Chapter FIFTEEN
Data Search

Exam Objectives

Search for data by using search methods of the Stream classes including findFirst, findAny, anyMatch, allMatch, noneMatch.

Finding and Matching

Searching is a common operation when you have a set of data.

The Stream API has two types of operation for searching.

Methods starting with *Find*:

```
Optional<T> findAny()
Optional<T> findFirst()
```

That search for an element in a stream. Since there's a possibility that an element couldn't be found (if the stream is empty, for example), the return type of this methods is an `Optional`.

And method ending with *Match*:

```
boolean allMatch(Predicate<? super T> predicate)
boolean anyMatch(Predicate<? super T> predicate)
boolean noneMatch(Predicate<? super T> predicate)
```

That indicate if a certain element matches the given predicate, that's why they return a `boolean`.

Since all these methods return a type different than a stream, they are considered **TERMINAL** operations.

findAny() and findFirst()

`findAny()` and `findFirst()` practically do the same, they return the first element they find in a stream:

```
IntStream stream = IntStream.of(1, 2, 3, 4, 5, 6, 7);
stream.findFirst()
    .ifPresent(System.out::println); // 1

IntStream stream2 = IntStream.of(1, 2, 3, 4, 5, 6, 7);
stream2.findAny()
    .ifPresent(System.out::println); // 1
```

If the stream is empty, they return an empty `Optional`:

```
Stream<String> stream = Stream.empty();
System.out.println(
    stream.findAny().isPresent()
); // false
```

Of course, you can combine these methods with other stream operations:

```
IntStream stream = IntStream.of(1, 2, 3, 4, 5, 6, 7);
stream
    .filter(i -> i > 4)
    .findFirst()
    .ifPresent(System.out::println); // 5
```

When to use `findAny()` and when to use `findFirst()`?

When working with parallel streams, it's harder to find the first element. In this case, it's better to use `findAny()` if you don't really mind which element is returned.

anyMatch(), allMatch(), and noneMatch()

`anyMatch()` returns true if any of the elements in a stream matches the given predicate:

```
IntStream stream = IntStream.of(1, 2, 3, 4, 5, 6, 7);
System.out.println(
    stream.anyMatch(i -> i%3 == 0)
); // true
```

If the stream is empty or if there's no matching element, this method returns `false`:

```
IntStream stream = IntStream.empty();
System.out.println(
    stream.anyMatch(i -> i%3 == 0)
); // false

IntStream stream2 = IntStream.of(1, 2, 3, 4, 5, 6, 7);
System.out.println(
    stream2.anyMatch(i -> i%10 == 0)
); // false
```

`allMatch()` returns true only if **ALL** elements in the stream match the given predicate:

```
IntStream stream = IntStream.of(1, 2, 3, 4, 5, 6, 7);
System.out.println(
    stream.allMatch(i -> i > 0)
); // true

IntStream stream2 = IntStream.of(1, 2, 3, 4, 5, 6, 7);
System.out.println(
    stream2.allMatch(i -> i%3 == 0)
); // false
```

If the stream is empty, this method returns **TRUE** without evaluating the predicate:

```
IntStream stream = IntStream.empty();
System.out.println(
    stream.allMatch(i -> i%3 == 0)
); // true
```

`noneMatch()` is the opposite of `allMatch()`, it returns true if **NONE** of the elements in the stream match the given predicate:

```
IntStream stream = IntStream.of(1, 2, 3, 4, 5, 6, 7);
System.out.println(
    stream.noneMatch(i -> i > 0)
); // false

IntStream stream2 = IntStream.of(1, 2, 3, 4, 5, 6, 7);
System.out.println(
    stream2.noneMatch(i -> i%3 == 0)
); // false

IntStream stream3 = IntStream.of(1, 2, 3, 4, 5, 6, 7);
System.out.println(
    stream3.noneMatch(i -> i > 10)
); // true
```

If the stream is empty, this method returns also **TRUE** without evaluating the predicate:

```
IntStream stream = IntStream.empty();
System.out.println(
    stream.noneMatch(i -> i%3 == 0)
); // true
```

Short-circuiting

All of these operations use something similar to the short-circuiting of `&&` and `||` operators.

Short-circuiting means that the evaluation stops once a result is found.

In the case of the *find** operations, it's obvious that they stop at the first found element.

But in the case of the **Match* operations, think about it, why would you evaluate all the elements of a stream when by evaluating the third element (for example) you can know if all or none (for example) of the elements will match?

Consider this code:

```
IntStream stream = IntStream.of(1, 2, 3, 4, 5, 6, 7);
boolean b = stream
                .filter(i -> {
                    System.out.println("Filter:" + i);
                    return i % 2 == 0; })
                .allMatch(i -> {
                    System.out.println("AllMatch:" + i);
                    return i < 3;
                });
System.out.println(b);
```

What would you think the output will be?

The output:

```
Filter:1
Filter:2
AllMatch:2
Filter:3
Filter:4
AllMatch:4
false
```

As you can see, first of all, operations on a stream are not evaluated sequentially (in this case, first filter all the elements and then evaluate if all elements match the predicate of `allMatch()`).

Second, we can see that as soon as an element passes the filter predicate (like `2`) the predicate of `allMatch()` is evaluated.

Finally, we can see short-circuiting in action. As soon as the predicate of `allMatch()` finds an element that doesn't evaluate to true (like `4`), the two stream operations are canceled, no further elements are processed and the result is returned.

Just remember:

- With some operations, the whole stream doesn't need to be processed.
- Stream operations are not performed sequentially.

Key Points

- The Stream API has two types of operation for searching. Methods starting with *Find*:
  ```
  Optional<T> findAny()
  Optional<T> findFirst()
  ```
- And method ending with *Match*:
  ```
  boolean allMatch(Predicate<? super T> predicate)
  boolean anyMatch(Predicate<? super T> predicate)
  boolean noneMatch(Predicate<? super T> predicate)
  ```
- Both types are considered **TERMINAL** operations.
- `findAny()` and `findFirst()` practically do the same, they return the first element they find in a stream. If the stream is empty, they return an empty `Optional`.
- When working with parallel streams, it's harder to find the first element, so in this case, it's better to use `findAny()` if you don't really mind which element is returned.
- `anyMatch()` returns true if any of the elements in a stream matches the given predicate. If the stream is empty or if there's no matching element, it returns `false`.
- `allMatch()` returns true only if **ALL** elements in the stream match the given predicate.
- `noneMatch()` returns true if **NONE** of the elements in the stream match the given predicate.
- Both `allMatch()` and `noneMatch()` return `true` if the stream is empty.
- All of these operations are short-circuiting, meaning that the evaluation stops once a result is found.

Self Test

1. Given:

```
public class Question_15_1 {
    public static void main(String[] args) {
        Stream<Integer> s = Stream.of(100, 45, 98, 33);
        s.anyMatch(i -> i > 50)
            .findAny()
            .ifPresent(System.out::println);
    }
}
```

What is the result?
A. `100`
B. `98`
C. Nothing is printed
D. Compilation fails

2. Which of the following methods of the Stream interface returns an `Optional` type?
A. `filter()`
B. `findMatch()`
C. `findAny()`
D. `anyMatch()`

3. Given:

```
public class Question_15_3 {
    public static void main(String[] args) {
        IntStream stream = IntStream.of(1, 2, 3, 4, 5, 6, 7);
        stream.allMatch(i -> {
            System.out.print(i);
            return i % 3 == 0;
        });
    }
}
```

What is the result?
A. `1234567`
B. `36`
C. `1`
D. Compilation fails

4. Given:

```
public class Question_15_4 {
    public static void main(String[] args) {
        IntStream stream = IntStream.of(1, 2, 3, 4, 5, 6, 7);
        stream.filter(i -> {
                return i > 3;
        }).anyMatch(i -> {
                System.out.print(i);
                return i % 2 == 1;
        });
    }
}
```

What is the result?
A. `45`
B. `5`
C. `4567`
D. Compilation fails

Open answers page

Do you like what you read? Would you consider?

Buying the print/kindle version from Amazon

Buying the PDF/EPUB/MOBI versions from Leanpub

Buying the e-book version from iTunes

Buying the e-book version from Kobo

Buying the e-book version from Scribd

Do you have a problem or something to say?

Report an issue with the book

Contact me

14. Optional Class 16. Stream Operations on Collections